PART. I
Django MVT

bstorm
TRAINING

1. Introduction à Django

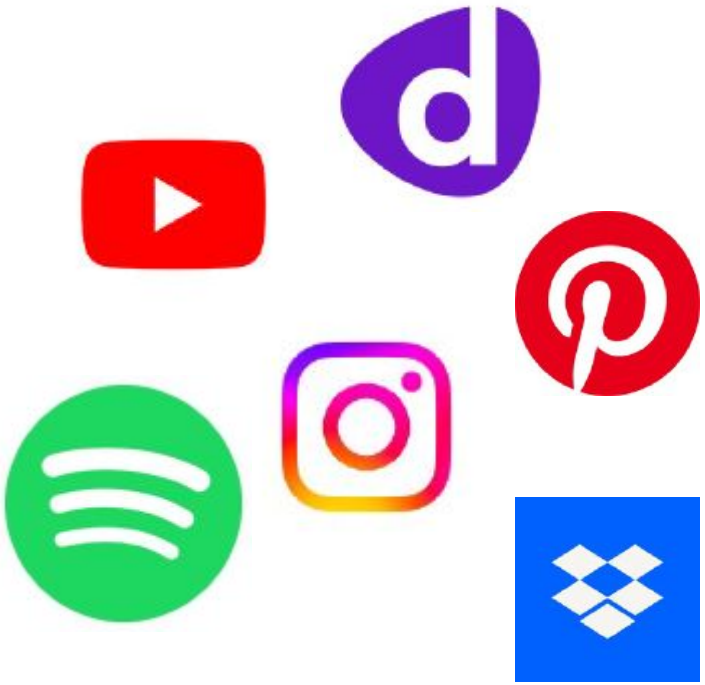
bstorm
TRAINING

Introduction à Django

Django, est un framework web de haut niveau écrit en Python. Il est conçu pour encourager le développement rapide de sites web sécurisés et maintenables sans réinventer la roue.

- Rapidité: Développez votre application web rapidement sans compromettre la qualité.
- Sécurité: Bénéficiez de fonctionnalités de sécurité intégrées, protégeant votre application contre les menaces courantes.
- Scalabilité: Django supporte la croissance d'une application, allant de quelques utilisateurs à des millions.

Ils utilisent Django



Pourquoi utiliser Django ?

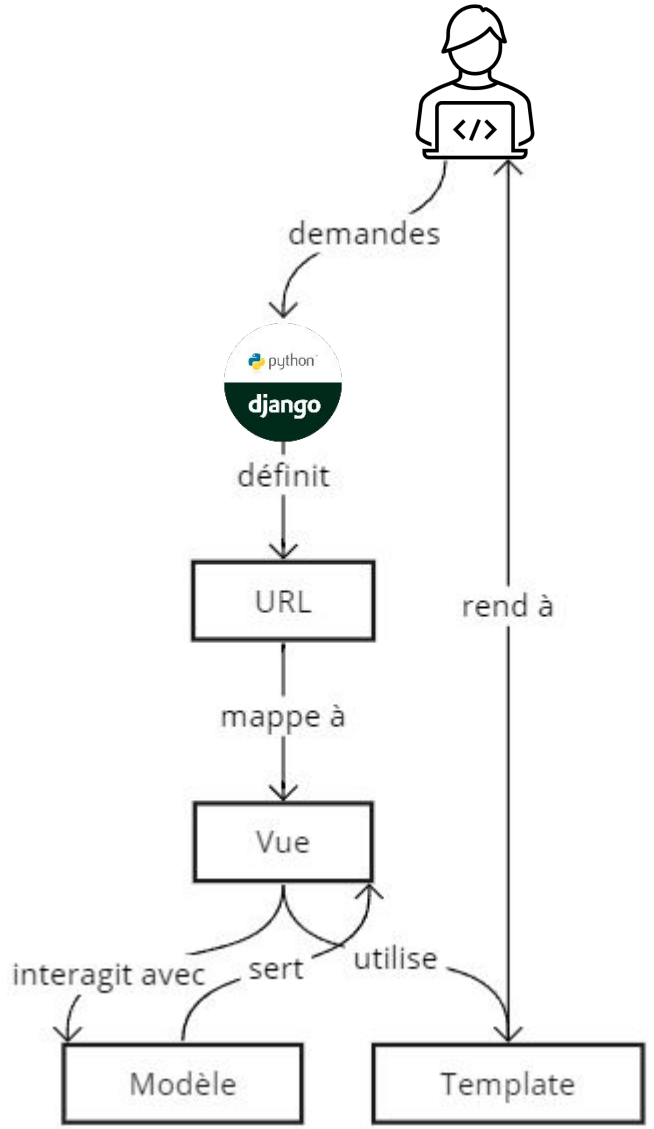
- **Productivité accrue** : Django facilite le développement en fournissant des fonctionnalités prêtes à l'emploi.
- **Sécurité** : Django intègre des mesures de sécurité par défaut pour protéger les applications Web.
- **Scalabilité** : Les applications Django peuvent facilement s'adapter à une augmentation de la charge de travail.
- **Communauté active** : Django bénéficie d'une grande communauté de développeurs et de contributeurs qui fournissent un support et des ressources.
- **Administration automatique** : L'interface d'administration de Django permet la gestion simple et rapide des données, offrant un gain de temps appréciable aux développeurs.

Architecture MVT de Django

- **Le Modèle**
  Il gère les données de l'application, définit leur structure et interagit avec la base de données pour stockage et récupération.

- **La Vue**
  Elle contrôle le flux d'informations en traitant les requêtes des utilisateurs, interagissant avec le modèle pour obtenir les données nécessaires.

- **Le Template**
  Il gère la présentation visuelle des données en utilisant HTML pour créer l'interface utilisateur, facilitant ainsi la séparation des logiques de présentation et de données.



2. Installation et Configuration

bstorm TRAINING

Installer Django

- Nous devons installer le Framework Django à l'aide de pip :

```
pip install django
```

- Nous pouvons ensuite nous assurer que l'installation est réussie :

```
django-admin --version
```

Création d'un Projet Django

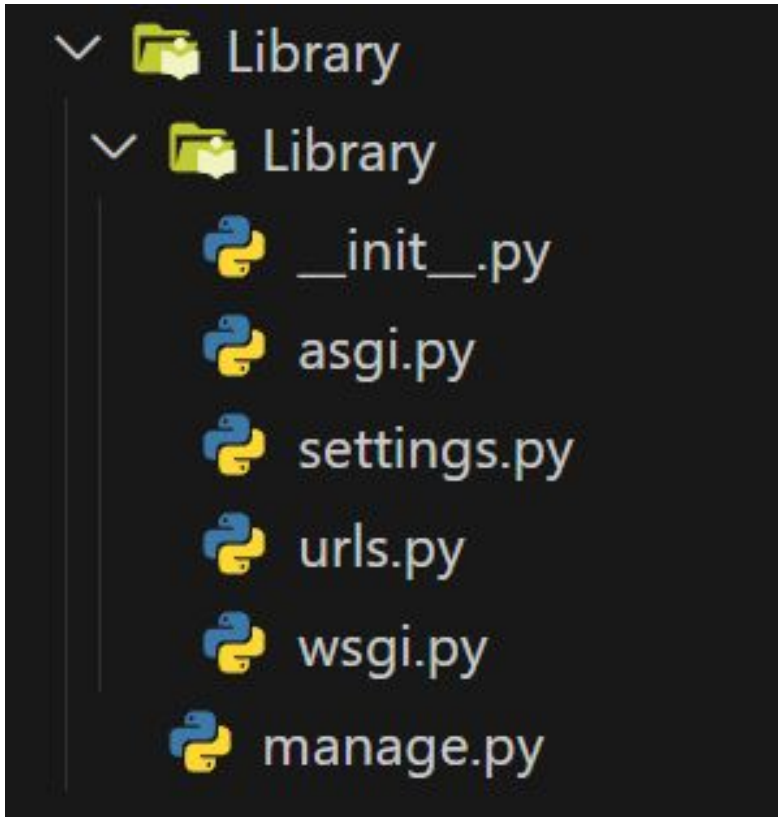
- Nous pouvons dès à présent créer un nouveau projet Django:

```
django-admin startproject monprojet
```

- Cette commande crée un nouveau dossier monprojet contenant la structure de base du projet Django, y compris le fichier de configuration settings.py, et un script de gestion manage.py

Exploration de la Structure de Projet

- Un projet Django fraîchement créé contient les éléments suivants :
  - monprojet/ : Le dossier racine de notre projet.
  - manage.py : Un script de ligne de commande qui vous aide à gérer notre projet.
  - monprojet/settings.py : Le fichier de configuration de notre projet Django.
  - monprojet/urls.py : Le fichier de déclaration des URLs de notre projet.



Exercice

Installer Django et créer un nouveau projet nommé MonBlog.

3. Les Applications

bstorm TRAINING

Introduction aux Applications Django

- Django encourage le développement modulaire à travers l'utilisation d'applications. Chaque application Django est un paquet Python qui représente une fonctionnalité ou un aspect spécifique de notre projet.

- Une bonne pratique consiste à rendre chaque application aussi indépendante et réutilisable que possible.

Créer une Nouvelle Application

Pour commencer avec une nouvelle application dans notre projet Django, utilisez la commande startapp.

```
python manage.py startapp books
```

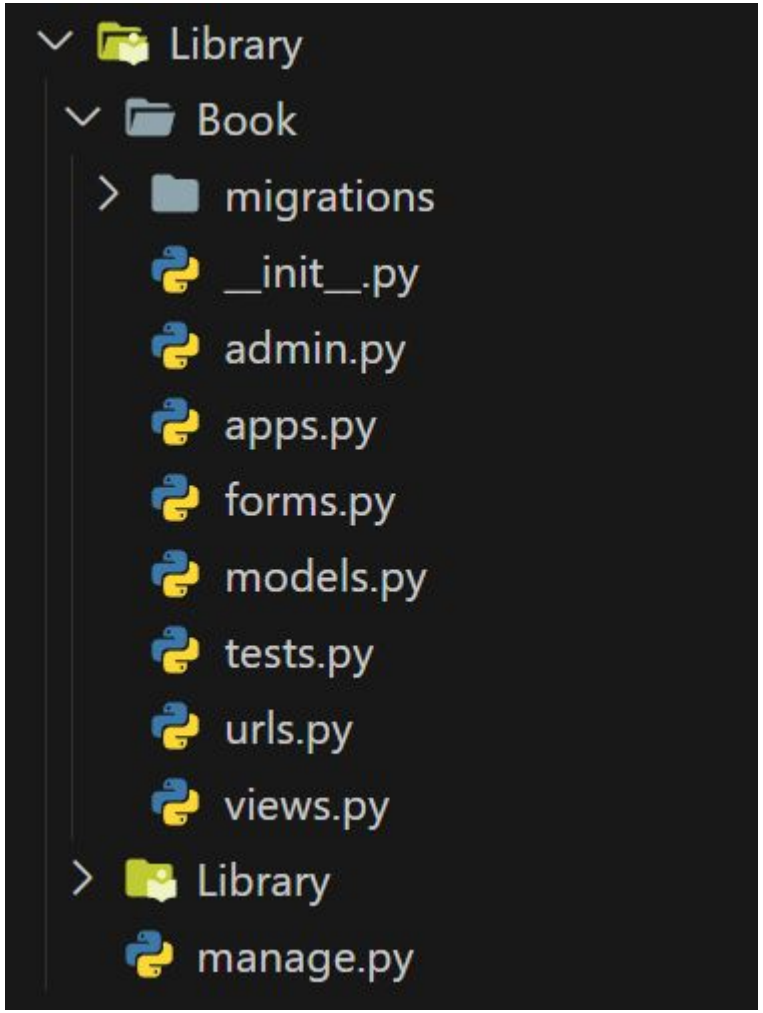
Cette commande crée une nouvelle application nommée books avec une structure de répertoire standard pour les fichiers de modèles, de vues, d'URLs, et autres.

Structure d'une Application Django

Chaque application aura sa structure contenant les bases du modèle MVT.

Nous y mettrons donc, selon nos besoins :

- Les Urls
- Les Vues
- Les Modèles
- Les Templates
- Les formulaires



Le Fichier apps.py

Chaque application Django contient un fichier apps.py qui définit la configuration spécifique de l'application.

```
from django.apps import AppConfig

class BooksConfig(AppConfig):
    name = 'books'
    verbose_name = 'Book Management'
```

Cette classe de configuration permet de nommer l'application et de définir un nom affiché plus convivial.

Exercice

Créez une application Django nommée article dans votre projet MonBlog.

4. Les Vues et URLs dans Django

bstorm TRAINING

Configurer les URLs dans Django

Les URLs, jouent un rôle crucial dans les applications web Django en définissant la manière dont les requêtes HTTP sont acheminées vers leurs vues correspondantes.
Chaque URL est mappée à une vue spécifique, qui traite la requête et retourne une réponse.

Pour ce faire, Django utilise un fichier urls.py à la racine du projet et dans chaque application pour déclarer les patterns d'URLs.
Ce système permet de modulariser et d'organiser efficacement les routes de l'application.

Django permet également d'inclure des paramètres dynamiques dans les URLs en utilisant des convertisseurs de type comme <int:id>, qui capturent des parties de l'URL et les passent en tant qu'arguments à la vue.
Cela permet de construire des applications web dynamiques capables de traiter des données spécifiques basées sur l'URL visitée par l'utilisateur.

Utilisation de la fonction ‘Path’

La fonction path() est utilisée pour associer un pattern d'URL à une vue.
Elle prend en premier argument une chaîne de caractères définissant le pattern d'URL, et en second argument la vue associée à ce pattern.
Le paramètre name optionnel permet de nommer l'URL, facilitant sa référence dans les templates et les vues.

Dans cet exemple, trois URLs sont définies :

Une URL racine qui pointe vers la vue accueil.

Une URL livres/ qui liste tous les livres via la vue liste_livres.

Une URL dynamique livres/<int:id>/ qui affiche les détails d'un livre spécifique via detail_livre, en utilisant un paramètre id pour identifier le livre.

```
from django.urls import path
from . import views
urlpatterns = [
    path('', views.accueil, name='accueil'),
    path('livres/', views.liste_livres, name='liste_livres'),
    path('livres/<int:id>/', views.detail_livre, name='detail_livre'),
]
```

Inclure les URLs d'une Application

Pour connecter les URLs d'une application au projet principal, utilisez la fonction include() dans le fichier urls.py du projet.

```
from django.urls import path, include

urlpatterns = [
    path('books/', include('books.urls')),
]
```

Cette configuration dirige toutes les requêtes commençant par books/ vers les URLs définies dans l'application books.

Vue de Django

- La Vue dans Django sont des fonctions Python qui prennent des données et retournent des résultats (HTML, JSON, etc.), simplifiant ainsi le processus de développement web.

- En définissant une vue dans Django, vous pouvez spécifier comment afficher les données des utilisateurs inscrits dans l'application.

- Les vues en Django organisent efficacement la logique métier, assurant ainsi une maintenabilité et une extensibilité optimales de l'application.

- Les vues en Django sont appelées sur base du fichier URLs.py

```python
from django.shortcuts import render
from .models import Livre
def livres_utilisateur(request,
utilisateur_id):
    utilisateur =
Utilisateur.objects.get(id=utilisateur_id)
    livres =
Livre.objects.filter(utilisateur=utilisateur)
    return render(request,
'bibliotheque/livres_utilisateur.html',
{'livres': livres})
```

Les Types de Réponses

Les vues Django renvoient une réponse au client. Cette réponse peut varier en type, en fonction des données à renvoyer et de la manière dont elles doivent être traitées par le client.

HttpResponse:
Le type de réponse le plus basique. Utilisé pour renvoyer du contenu HTML ou du texte.

```python
from django.http import HttpResponse
def ma_vue(request):
    return HttpResponse("Voici une réponse simple.")
```

JsonResponse:
Spécialisé pour envoyer des réponses JSON. Utile pour les API REST ou les réponses AJAX

```python
from django.http import JsonResponse
def ma_vue(request):
    data = {"nom": "Django", "type": "Framework"}
    return JsonResponse(data)
```

Redirect:
Pour rediriger l'utilisateur vers une autre URL

```python
from django.shortcuts import redirect
def ma_vue(request):
    return redirect('/une-autre-vue/')
```

Render:
Utilisé pour générer une réponse en combinant un template avec un dictionnaire de contexte.
Avantages: Permet de créer des réponses HTML riches en intégrant des données dynamiques.

```
from django.shortcuts import render
def liste_livres(request):
    livres = Livre.objects.all()
    return render(request, 'livres/liste.html', {'livres': livres})
```

Nous verrons les Templates juste après, mais dans cet exemple, render prend la requête HTTP, le chemin du template ('livres/liste.html'), et un contexte contenant une liste de tous les livres. Le moteur de template de Django rend le template avec le contexte fourni, générant ainsi une page HTML à envoyer au client.

Le paramètre ‘Request’

Depuis le début de l’ utilise des vues, nous utilisons le paramètre ‘request’, mais à quoi sert-il ?

L'objet request est un élément central dans le traitement d'une vue Django. Il contient des informations sur la requête HTTP effectuée par le client.

request.method: La méthode HTTP utilisée pour la requête (par exemple, 'GET' ou 'POST’).

request.GET: Un dictionnaire-like contenant tous les paramètres GET. Utilisé pour accéder aux données envoyées dans l'URL.

request.POST: Similaire à request.GET, mais pour les données envoyées via des formulaires HTML en utilisant la méthode POST.

request.FILES: Contient tous les fichiers téléchargés avec la requête, accessible lorsque la méthode POST est utilisée et le formulaire contient enctype="multipart/form-data".

request.user: L'utilisateur effectuant la requête, utile pour les fonctionnalités d'authentification et de permission.

Les paramètres dynamiques

Les vues Django peuvent également recevoir des informations directement à travers l'URL. Cela est souvent utilisé pour capturer des parties de l'URL comme des variables.

Exemple d'URL avec récupération de l'id:

```
from django.urls import path
from . import views
urlpatterns = [
    path('livres/<int:id>/', views.detail_livre, name='detail_livre'),
]
```

```
from django.shortcuts import render, get_object_or_404
from .models import Livre
def detail_livre(request, id):
    livre = get_object_or_404(Livre, pk=id)
    return render(request, 'livres/detail_livre.html', {'livre': livre})
```

Exercice

Créez une vue simple et paramétrer vos URLs pour afficher "Bonjour, bienvenue sur MonBlog !" sur la page d'accueil.

5. Les Templates Django

bstorm
TRAINING

Template de Django

- Les Templates dans Django permettent de séparer le design de la logique métier, simplifiant ainsi la maintenance et l'évolutivité du projet.

- En Django, les templates sont des fichiers HTML contenant des balises spéciales pour l'ajout de données dynamiques, assurant une présentation cohérente.

- L'utilisation efficace des templates dans Django garantit une expérience utilisateur harmonieuse en affichant les données de manière structurée et attrayante.

```
<!DOCTYPE html>
<html>
<head>
    <title>Livres de {{ utilisateur.nom }}</title>
</head>
<body>
    <h1>Livres de {{ utilisateur.nom }}</h1>
    <ul>
        {% for livre in livres %}
            <li>{{ livre.titre }} par {{ livre.auteur }}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

Syntaxe des Templates Django

- Dans les templates Django, la syntaxe de base comprend l'interpolation de variables et la logique de template.
- Les variables sont représentées par {{ variable }} et la logique de template par {% expression %}.

```
<p>Le livre sélectionné est : {{ book.title }}</p>
```

Les Tags de Template

Les tags de template sont des constructions qui contrôlent la logique et le flux de la présentation. Ils sont encadrés par {% et %}.

Liste de Tags Communs:

- **{% if %}...{% endif %}** pour les conditions.
- **{% for %}...{% endfor %}** pour les boucles.
- **{% block %}...{% endblock %}** pour définir des zones surchargeables.
- **{% include %}** pour inclure d'autres templates.
- **{% extends %}** pour l'héritage de templates.

Exemple de Code pour la Boucle:

```
<ul>
{% for author in book.authors %}
  <li>{{ author.name }}</li>
{% endfor %}
</ul>
```

Exemple de Code pour une Condition:

```
{% if book.is_available %}
  <p>Ce livre est disponible.</p>
{% else %}
  <p>Ce livre n'est pas disponible actuellement.</p>
{% endif %}
```

Les Filtres de Template

Les filtres modifient la façon dont les variables sont affichées. Ils sont utilisés avec {{ variable|filter }}.

Liste de Filtres Communs:

date : pour formater une date selon le format spécifié.
length : pour obtenir la longueur d'une liste ou d'une chaîne de caractères.
default : pour retourner une valeur par défaut si la variable est nulle ou vide.
lower : pour convertir une chaîne de caractères en minuscules.
upper : pour convertir une chaîne de caractères en majuscules.
linebreaksbr : pour remplacer les sauts de ligne par des balises
 HTML.
slice : pour sélectionner une plage d'éléments dans une liste.
pluralize : pour sélectionner le pluriel d'un mot en fonction d'une variable.
urlencode : pour encoder une chaîne de caractères pour une utilisation dans une URL.

Exemples :

```
<p>Date de publication : {{ book.publish_date|date:"Y-m-d" }}</p>

<p>Longueur de la liste : {{ my_list|length }}</p>

<p>Valeur par défaut : {{ my_variable|default:"N/A" }}</p>

<p>Texte en minuscules : {{ my_string|lower }}</p>

<p>Texte en majuscules : {{ my_string|upper }}</p>

<p>Texte avec sauts de ligne HTML : {{ my_text|linebreaksbr }}</p>

{% for item in my_list|slice:"2:5" %}
  <p>{{ item }}</p>
{% endfor %}

<p>{{ count }} message{{ count|pluralize }}</p>

<p>Chaîne de caractères encodée pour URL : {{ my_string|urlencode }}</p>
```

Héritage de Templates

L'héritage permet de définir un template de base qui contient la structure générale du site que les autres templates peuvent étendre et surcharger.

Exemple de Code pour le Template de Base:

```
<!-- base.html -->
<html>
<head>
    <title>{% block title %}Ma Bibliothèque{% endblock %}</title>
</head>
<body>
    {% block content %}{% endblock %}
</body>
</html>
```

Exemple de Code pour le Template de Base:

```
{% extends "base.html" %}
{% block title %}Page de Détail du Livre{% endblock %}
{% block content %}
    <h2>{{ book.title }}</h2>
    <p>Par {{ book.author }}</p>
{% endblock %}
```

Les Blocs dans les Templates Django

Les blocs sont des sections définies dans un template parent qui peuvent être surchargées par les templates enfants grâce à l'héritage.

```
<!-- Dans base.html -->
{% block navbar %}...{% endblock navbar %}
{% block content %}...{% endblock content %}

<!-- Dans un autre template qui étend
base.html -->
{% extends "base.html" %}
{% block navbar %}
    <nav>Ma barre de navigation
surchargée</nav>
{% endblock navbar %}
```

Inclusion de Templates

L'inclusion est un moyen d'intégrer le contenu d'un template dans un autre sans héritage.

```
<div id="sidebar">
    {% include "sidebar.html" %}
</div>
```

Contrairement à l'héritage, l'inclusion n'établit pas de relation parent-enfant entre les templates. Elle permet simplement d'insérer un template dans un autre, ce qui est utile pour des éléments réutilisables comme des barres latérales, des en-têtes ou des pieds de page.

Exercice

Créez un template pour la page d'accueil qui étend un template de base et affiche un message de bienvenue.

6. Les Modèles Django

bstorm
TRAINING

Modèle de Django

- Le Modèle de Django définit la structure des données avec des champs comme 'titre', 'contenu' et 'date de publication.
- Il interagit avec la base de données pour gérer la persistance des informations liées à l'application web.
- En associant chaque modèle à une table de base de données, Django facilite la manipulation et l'organisation des données dans l'application.

```python
from django.db import models
class Livre(models.Model):
    titre = models.CharField(max_length=100)
    auteur = models.CharField(max_length=100)
    date_lecture = models.DateField(null=True, blank=True)
    note = models.IntegerField(null=True, blank=True)
```

Les Types de champs

Type de Champ	Description
CharField	Un champ pour stocker des chaînes de caractères de longueur limitée. Requiert l'argument max_length.
TextField	Un champ pour stocker de longs textes sans limite de longueur.
IntegerField	Un champ pour stocker des nombres entiers.
DecimalField	Un champ pour stocker des nombres décimaux. Requiert les arguments max_digits et decimal_places.
FloatField	Un champ pour stocker des nombres flottants avec des valeurs à virgule flottante.
BooleanField	Un champ pour stocker des valeurs booléennes (True ou False).
DateField	Un champ pour stocker des dates. Peut avoir l'argument auto_now_add pour enregistrer la date de création.
DateTimeField	Un champ pour stocker des dates et heures. Peut aussi utiliser auto_now_add.
EmailField	Un champ pour stocker des adresses email. Vérifie que la valeur saisie est une adresse email valide.
FileField	Un champ pour stocker un fichier téléchargé.
ImageField	Semblable à FileField, mais spécifiquement pour les images. Vérifie que le fichier téléchargé est une image.
ForeignKey	Un champ pour créer une relation "many-to-one". Requiert un argument spécifiant le modèle vers lequel il pointe.
ManyToManyField	Un champ pour créer une relation "many-to-many". Similaire à ForeignKey, mais permet des relations réciproques.
OneToOneField	Un champ pour créer une relation "one-to-one". Permet de lier un objet à un seul autre objet.
SlugField	Un champ pour stocker des "slugs" (une version URL-friendly d'un nom ou titre).
URLField	Un champ pour stocker des URL. Vérifie que la valeur saisie est une URL valide.

Relations entre les Modèles

Les modèles Django peuvent être liés les uns aux autres pour représenter des relations complexes comme dans les bases de données relationnelles.

ForeignKey - Relation "Many-to-One":

Utilisé pour créer une relation "many-to-one". Chaque instance du modèle peut être associée à une instance d'un autre modèle.

ManyToManyField - Relation "Many-to-Many":

Permet de créer une relation "many-to-many" entre deux modèles.

OneToOneField - Relation "One-to-One":

Utilisé pour créer une relation "one-to-one" entre deux modèles.

ForeignKey - Relation "Many-to-One":

Exemple : un modèle Commentaire qui se rapporte à un Livre, permettant à chaque livre d'avoir de nombreux commentaires.

```
class Commentaire(models.Model):
    livre = models.ForeignKey(Livre, on_delete=models.CASCADE)
    texte = models.TextField()
```

ManyToManyField - Relation "Many-to-Many":

Exemple : un modèle Auteur lié à un modèle Livre, où un livre peut avoir plusieurs auteurs et un auteur peut écrire plusieurs livres.

```
class Auteur(models.Model):
    nom = models.CharField(max_length=100)
    livres = models.ManyToManyField(Livre)
```

OneToOneField - Relation "One-to-One":

Exemple : un modèle DétailsLivre qui stocke des informations supplémentaires pour un Livre. Chaque livre ne peut avoir qu'un seul ensemble de détails.

```
class DetailsLivre(models.Model):
    livre = models.OneToOneField(Livre, on_delete=models.CASCADE)
    resume = models.TextField()
```

Configurer la base de données

Django est livré avec une base de données SQLite par défaut, qui est suffisante pour les projets de petite à moyenne taille et pour l'apprentissage. Assurez-vous que les configurations de la base de données dans settings.py sont correctes. Si vous utilisez SQLite, la configuration par défaut devrait fonctionner sans problème.
Dans notre cas, nous allons utiliser PostegreSQL.

Pour ce faire nous avons besoin d'installer un package :

```
pip install psycopg2
```

Puis, configurez les paramètres de connexion à la base de données dans le fichier settings.py :

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'nom_de_votre_base_de_données',
        'USER': 'votre_utilisateur',
        'PASSWORD': 'votre_mot_de_passe',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

Finalement, il nous reste à effectuer une migration afin d'appliquer les changements :

```
python manage.py migrate
```

Migration de notre Modèle

Les migrations sont le moyen de Django de propager les changements que vous apportez à vos modèles (ajout de champs, changement de champs, etc.) dans notre schéma de base de données. Les migrations permettent de modifier la structure de notre base de données de manière contrôlée et versionnée sans perdre de données.

Création de Migrations:
Après avoir défini ou modifié vos modèles, utilisez la commande makemigrations pour créer des fichiers de migration.

```
python manage.py makemigrations
```

Application des Migrations:
Utilisez la commande migrate pour appliquer les migrations générées à la base de données, effectuant ainsi les modifications du schéma.

```
python manage.py migrate
```

Exercice

Mettez en place une database appelée ‘mvt_blog’ dans PostgreSQL

Mettez en place les paramètres d’utilisation de PostgreSQL

Définissez un modèle Article avec des champs pour un titre(varchar), un auteur(varchar), et le contenu(text) d'un article (faire la migration).

Effectuez les migrations vers la base de donnée

7. L'administration Django

bstorm TRAINING

Introduction au Panneau d'Administration Django

Le panneau d'administration de Django est une fonctionnalité puissante et intégrée qui illustre la flexibilité du modèle MVT (Modèle-Vue-Template) de Django. Voici quelques points clés qui soulignent son intérêt :

- **Accès rapide à la gestion des données** : Permet aux développeurs et administrateurs de visualiser, ajouter, modifier et supprimer les données stockées dans les modèles, directement via une interface web conviviale.
- **Facile à configurer** : Peut être mis en place avec très peu de code supplémentaire, Django générant automatiquement l'interface pour les modèles enregistrés.
- **Hautement personnalisable** : Bien que très fonctionnel dès son installation, le panneau d'administration peut être étendu et personnalisé pour s'adapter aux besoins spécifiques du projet.
- **Sécurisé** : Intègre des mécanismes d'authentification et de permissions, assurant que seuls les utilisateurs autorisés peuvent accéder à certaines données ou fonctionnalités.

Accès au Panneau d'Administration

- Pour accéder au panneau d'administration, vous devez d'abord créer un superutilisateur à l'aide de la commande :

```
python manage.py createsuperuser
```

- Après avoir créé un superutilisateur, vous pouvez vous connecter au panneau d'administration en visitant /admin sur notre site.

Ajouter un Modèle au Panneau d'Administration

Pour ajouter un modèle au panneau d'administration et le rendre gérable via l'interface, vous devez l'enregistrer dans le fichier admin.py de notre application.

```
from django.contrib import admin
from .models import Book

admin.site.register(Book)
```

Cet exemple montre comment enregistrer le modèle *Book* pour qu'il apparaisse dans le panneau d'administration.

Exercice

Créez un superutilisateur et utilisez l'interface d'administration Django pour gérer les articles.

Personnalisez l'interface d'administration pour améliorer la gestion de votre modeles Article

8. Les Formulaires

bstorm TRAINING

Introduction aux Formulaires Django

- Les formulaires sont un moyen essentiel d'interagir avec les utilisateurs, permettant de collecter des données saisies par l'utilisateur.
- Django fournit un système puissant de formulaires qui facilite la création de formulaires HTML et la validation des données entrantes.
- Un formulaire Django peut être créé à partir d'une classe Form qui définit les champs du formulaire et leur comportement.

```python
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField(label='Votre nom', max_length=100)
    message = forms.CharField(widget=forms.Textarea)
```

- Il est également possible de créer un formulaire automatiquement sur base des champs d'un modèle existant.

```python
from django import forms
from .models import Book

class BookForm(forms.ModelForm):
    class Meta:
        model = Book
        fields = ['title', 'author', 'summary', 'genre']
```

Afficher un Formulaire dans un Template

Pour afficher un formulaire dans un template, passez une instance de BookForm à notre template via le contexte.

```
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Ajouter un Livre</button>
</form>
```

Traiter les Données du Formulaire dans une Vue

Les vues Django traitent les données du formulaire après la soumission. Pour cela, vérifiez la méthode de requête et utilisez la méthode is_valid() du formulaire pour valider les données.

```python
from django.shortcuts import render, redirect
from .forms import BookForm

def add_book(request):
    if request.method == 'POST':
        form = BookForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('books:list')
    else:
        form = BookForm()
    return render(request, 'books/add_book.html', {'form': form})
```

La vue add_book traite la requête POST, valide le formulaire, sauvegarde le nouveau livre dans la base de données et redirige l'utilisateur vers la liste des livres.

Personnalisation et Style des Formulaires

Les formulaires Django peuvent être personnalisés pour correspondre au style et aux besoins spécifiques de notre site. Cela inclut la modification des widgets, l'ajout de classes CSS et la personnalisation de la validation.

```
class BookForm(forms.ModelForm):
    class Meta:
        model = Book
        fields = ['title', 'author', 'summary', 'genre']
        widgets = {
            'summary': forms.Textarea(attrs={'cols': 80, 'rows': 20}),
        }
```

Dans cet exemple, le champ summary du formulaire utilise un widget Textarea avec des attributs CSS personnalisés pour la largeur et la hauteur.

Validation des Formulaires

La validation des données de formulaire est essentielle pour l'intégrité de l'application. Django fournit un système robuste de validation automatique et permet également de définir des méthodes de validation personnalisées.

```python
class BookForm(forms.ModelForm):
    # ... autres options et champs ...

    def clean_title(self):
        title = self.cleaned_data['title']
        if 'Django' in title:
            raise forms.ValidationError("Le titre ne peut pas contenir 'Django'.")
        return title
```

Ici, une méthode clean_title() personnalisée est ajoutée à BookForm pour valider que le titre du livre ne contient pas le mot 'Django'.

Personnaliser les formulaires dans le panneau Admin

Vous pouvez personnaliser la manière dont vos modèles sont affichés dans le panneau d'administration en définissant une classe ModelAdmin.

```
from django.contrib import admin
from .models import Book

class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'author', 'genre')
    list_filter = ('genre',)
    search_fields = ('title', 'author__name')

admin.site.register(Book, BookAdmin)
```

Ce code configure l'affichage de la liste des livres, ajoute un filtre par genre et permet la recherche par titre et nom d'auteur.

Configurer les Formulaires d'Administration

Pour une personnalisation plus avancée, vous pouvez spécifier des formulaires personnalisés pour l'ajout ou la modification des instances de modèles dans l'administration.

```python
from django import forms
from django.contrib import admin
from .models import Book

class BookForm(forms.ModelForm):
    class Meta:
        model = Book
        fields = ['title', 'author', 'summary', 'genre']

class BookAdmin(admin.ModelAdmin):
    form = BookForm

admin.site.register(Book, BookAdmin)
```

Cet exemple utilise un formulaire personnalisé pour l'édition des livres dans l'administration.

Exercice

Créez un formulaire Django pour soumettre de nouveaux Articles dans votre Blog

9. L'ORM Django

bstorm
TRAINING

Introduction à l'ORM Django

L'ORM (Object-Relational Mapping) de Django est une couche d'abstraction qui facilite l'interaction avec la base de données en utilisant du code Python au lieu de SQL brut.

Les opérations CRUD :

- Créer
- Lire
- Mettre à jour
- Supprimer

Ce sont les quatre opérations de base pour interagir avec les données.

Créer des Données avec l'ORM

Pour créer de nouvelles entrées dans la base de données, utilisez les classes de modèle Django et la méthode save().

```
from books.models import Book

new_book = Book(title='Apprendre Django', author='A. Developer', genre='Éducatif')
new_book.save()
```

Ce code crée une nouvelle instance du modèle Book et l'enregistre dans la base de données.

Lire des Données avec l'ORM

Django permet de lire des données à l'aide de méthodes telles que all(), get(), et filter().

- Récupération de Tous les Livres:

```
all_books = Book.objects.all()
```

- Récupération d'un Livre Spécifique:

```
specific_book = Book.objects.get(id=1)
```

Mettre à Jour des Données avec l'ORM

```
def book_update(request, id):
    book_from_db = Book.objects.get(id=id)
    #POST
    if request.method == 'POST':
        form = Create_books_form(request.POST, instance=book_from_db)
        if form.is_valid():
            form.save()
            return redirect('books_list')
    #GET
    form_from_view = Create_books_form(instance= book_from_db)
    return render(request, 'books/book_update.html', {'form_in_template' : form_from_view}
)
```

Cet exemple montre comment changer le titre d'un livre et le sauvegarder.

Supprimer des Données avec l'ORM

La suppression de données se fait en récupérant l'instance à supprimer et en appelant la méthode delete()

```
book_to_delete = Book.objects.get(id=1)
book_to_delete.delete()
```

Ce code supprimera l'objet *Book* avec *l'id* spécifié de la base de données.

Requêtes Complexes avec l'ORM

L'ORM de Django permet d'effectuer des requêtes complexes, telles que des jointures, des agrégations, ou des filtres conditionnels complexes.

```
from django.db.models import Q

books = Book.objects.filter(Q(title__contains='Django') |
Q(author__name__contains='Developer'))
```

Q est un objet générique fourni par Django pour représenter des expressions de requête complexes. Il fait partie de l'ORM (Object-Relational Mapping) de Django et est utilisé pour construire des conditions de filtrage qui peuvent impliquer des opérateurs logiques comme AND, OR, et NOT.

Optimisation des Requêtes avec select_related et prefetch_related

- select_related est utilisé pour optimiser les requêtes "one-to-many" en réduisant le nombre de requêtes SQL avec une jointure SQL.

```
books_with_authors = Book.objects.select_related('author').all()
```

- prefetch_related est similaire mais pour les relations "many-to-many" ou "many-to-one" en faisant des requêtes séparées et en les combinant ensuite en Python.

```
from django.db.models import Prefetch

books_with_genres = Book.objects.prefetch_related('genres').all()
```

Exercice

Utilisez l'ORM Django pour créer votre CRUD des articles.

Implémentez les vues, les templates, les urls nécessaires pour :

- Afficher un article (titre, contenu, auteur)
- Afficher tous les articles (liste des titres)
- Mettre à jour un article
- Supprimer un article

Cliquer sur le titre d'un article dans l'affichage de tous les articles, renvoie vers l'article détaillé.

10. Les Fichiers Statiques et le Design

bstorm TRAINING

Introduction aux Fichiers Statiques

- Les fichiers statiques incluent les feuilles de style CSS, les scripts JavaScript et les images qui ne changent pas à chaque requête.
- Django facilite la gestion de ces ressources à travers le système de fichiers statiques.
- Avant la mise en production, il est possible de rassembler tous les fichiers statiques

Configuration de Base

- Pour utiliser des fichiers statiques dans Django, définissez `STATIC_URL` dans notre fichier `settings.py`, qui est le chemin d'URL utilisé pour référencer les fichiers statiques.

- Utilisez `STATICFILES_DIRS` pour spécifier les chemins du système de fichiers où Django doit rechercher les fichiers statiques.

```
STATIC_URL = '/static/'
STATICFILES_DIRS = [
    BASE_DIR / "static",
]
```

Utilisation des Fichiers Statiques dans les Templates

- Pour référencer des fichiers statiques dans vos templates, utilisez la balise de template {% static %}.

```
<link rel="stylesheet" href="{% static 'css/style.css' %}">
```

- Il faut également charger ce tag de template en début de page avec {% load static %}

```
{% load static %}
```

Une bonne pratique est d'organiser vos fichiers statiques par type (CSS, JS, images) et par application, pour maintenir la structure de notre projet claire et maintenable.

Déploiement des Fichiers Statiques

En production, utilisez la commande collectstatic de Django pour rassembler tous les fichiers statiques dans un seul répertoire défini par STATIC_ROOT.

```
python manage.py collectstatic
```

- **Optimisation des Fichiers Statiques**
  - Pour améliorer la performance, envisagez de minimiser vos fichiers CSS et JavaScript, et d'utiliser des outils ou des services pour compresser les images.
- **Utilisation de CDN pour les Fichiers Statiques**
  - Un réseau de distribution de contenu (CDN) peut être utilisé pour servir vos fichiers statiques, réduisant ainsi le temps de chargement des pages et améliorant l'expérience utilisateur.
- **Sécurité des Fichiers Statiques**
  - Assurez-vous que vos fichiers statiques ne contiennent pas de données sensibles ou de code susceptible d'être exploité.

Intégration des Frameworks Front-end

Django peut être facilement intégré avec des frameworks front-end populaires comme Bootstrap ou React pour accélérer le développement du design.

```
<link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css">
```

Exercice

Personnalisez le style de votre Blog :

- Ajouter un ou des fichiers CSS statiques
- Ajouter Bootstrap
- Modifier la couleur de fond de votre menu de navigation
- Modifier tous les liens (Voir article, Supprimer article, etc...) par des boutons Bootstrap

11. Authentification et Sécurité

bstorm TRAINING

Introduction à l'Authentification et la Sécurité dans Django

- Django fournit un système d'authentification robuste qui permet de gérer les utilisateurs, les sessions, les groupes, les permissions et plus encore.

- La sécurité est une priorité dans Django, avec des fonctionnalités intégrées pour protéger contre les vulnérabilités courantes telles que le Cross-Site Scripting (XSS), le Cross-Site Request Forgery (CSRF), et l'injection SQL.

Le Système d'Utilisateurs dans Django

- Django dispose d'un modèle User intégré pour l'authentification qui peut être étendu pour répondre à des besoins spécifiques.

- La création, la gestion et l'authentification des utilisateurs sont facilitées par des vues et des formulaires prédéfinis.

```
from django.contrib.auth.models import User

user = User.objects.create_user(username='johndoe', email='john@example.com',
password='password')
```

Le Système d'Utilisateurs dans Django

Django fournit des outils pour la gestion sécurisée des mots de passe, y compris le hachage, la vérification et la réinitialisation des mots de passe.

```python
from django.contrib.auth import authenticate

user = authenticate(username='johndoe', password='password')
if user is not None:
    # Authentification réussie
```

Gestion des Permissions

Dans le cadre du développement d'applications avec Django, la gestion des permissions est essentielle pour contrôler l'accès aux différentes fonctionnalités de notre site. Django fournit un système de permissions flexible, permettant de définir des accès basiques pour distinguer les utilisateurs connectés des visiteurs anonymes, ainsi que des permissions personnalisées pour des besoins spécifiques. Ce système assure que les actions sensibles, comme la modification ou la suppression de contenu, sont réservées aux utilisateurs autorisés, offrant un contrôle précis et sécurisé de notre application.

Django utilise des décorateurs pour restreindre l'accès aux vues en fonction de l'état d'authentification de l'utilisateur ou de ses permissions.

```python
from django.contrib.auth.decorators import login_required

@login_required
def my_view(request):
    # Vue accessible uniquement par les utilisateurs authentifiés
```

Gestion des Permissions

Créer une Permission Personnalisée

Ajoutez une permission personnalisée dans la classe Meta de notre modèle Book.

```python
class Book(models.Model):
    # attributs du modèle
    class Meta:
        permissions = [("update_book", "Can update book")]
```

Gestion des Permissions

Assigner la Permission à un Utilisateur via le Code

Importez les modèles nécessaires et récupérez l'utilisateur et la permission.

```python
from django.contrib.auth.models import User, Permission
from django.contrib.contenttypes.models import ContentType
from .models import Book

# Récupérer l'utilisateur et la permission
user = User.objects.get(username='username_de_l_utilisateur')
content_type = ContentType.objects.get_for_model(Book)
permission = Permission.objects.get(codename='update_book', content_type=content_type)

# Assigner la permission à l'utilisateur
user.user_permissions.add(permission)
```

Gestion des Permissions

Utiliser le décorateur @permission_required que nous fournit Django

le décorateur @permission_required est utilisé pour vérifier si l'utilisateur connecté a la permission library.update_book. Si l'utilisateur n'a pas cette permission, Django lèvera une exception PermissionDenied qui par défaut renvoie une réponse HTTP 403 Forbidden, à moins que raise_exception ne soit défini sur True, auquel cas l'utilisateur sera redirigé vers la page de connexion si non authentifié.

```python
from django.http import HttpResponseForbidden
from django.shortcuts import get_object_or_404, redirect, render
from .models import Book
from .forms import BookForm
from django.contrib.auth.decorators import permission_required

@permission_required('library.update_book', raise_exception=True)
def update_book(request, pk):
    book = get_object_or_404(Book, pk=pk)
    if request.method == 'POST':
        form = BookForm(request.POST, instance=book)
        if form.is_valid():
            form.save()
            return redirect('book_detail', pk=book.pk)
    else:
        form = BookForm(instance=book)
    return render(request, 'books/book_form.html', {'form': form})
```

Gestion des Sessions

Django MVT (Modèle-Vue-Template) offre un puissant système de gestion des sessions qui vous permet de stocker et de récupérer des données spécifiques à chaque utilisateur de manière sécurisée. Les sessions Django sont un moyen essentiel pour maintenir l'état entre les requêtes HTTP, qui sont par nature sans état.

```python
# Définir une valeur de session
request.session['favorite_book'] = 'Les Misérables'

# Accéder à une valeur de session
favorite_book = request.session.get('favorite_book', 'Aucun')
```

Protéger les Vues avec des Décorateurs

Django utilise des décorateurs pour restreindre l'accès aux vues en fonction de l'état d'authentification de l'utilisateur ou de ses permissions.

```python
from django.contrib.auth.decorators import login_required

@login_required
def my_view(request):
    # Vue accessible uniquement par les utilisateurs authentifiés
```

Protection contre le CSRF

Le token CSRF protège contre les attaques de type Cross-Site Request Forgery. Django l'intègre automatiquement dans les formulaires.

```
<form method="post">
    {% csrf_token %}
    <!-- Éléments du formulaire -->
</form>
```

Sécurité dans Django

- **Protection contre l'Injection SQL**

  L'utilisation de l'ORM Django et l'évitement de requêtes SQL brutes protègent naturellement contre les injections SQL.

- **Protection contre le XSS**

  Django échappe automatiquement les données sortantes dans les templates pour prévenir le Cross-Site Scripting (XSS)

- **Utilisation des Signaux pour la Sécurité**

  Les signaux peuvent être utilisés pour exécuter des actions de sécurité personnalisées, comme enregistrer des tentatives de connexion infructueuses ou changer des paramètres de sécurité en réponse à certains événements.

Bonnes Pratiques de Sécurité

- Toujours mettre à jour Django et les dépendances pour bénéficier des derniers correctifs de sécurité.
- Utiliser django-check pour identifier les problèmes de configuration de sécurité.
- Ne jamais stocker de données sensibles ou des mots de passe en clair dans la base de données.

La gestion des fichiers statiques dans Django est cruciale pour ajouter du style et de l'interactivité à vos applications web. Dans cette section, nous aborderons comment intégrer et optimiser l'utilisation des fichiers statiques pour le design de vos projets Django.

Exercice

Ajoutez une authentification utilisateur à votre blog.

Implémentez les fonctionnalités permettant aux utilisateurs de s'inscrire, de se connecter et de se déconnecter

PART. II
Django REST API

bstorm
TRAINING

1. Introduction aux API REST

bstorm TRAINING

Définition des API REST

- **Une API** (Application Programming Interface) est un ensemble de règles et de définitions qui permet à deux applications logicielles de communiquer entre elles. Dans le contexte du développement web, une API permet à des applications web et à des services d'échanger des données sur internet.

- **REST** (Representational State Transfer) est un style architectural qui définit un ensemble de contraintes pour la création d'interfaces web. Les API qui suivent les principes REST, souvent appelées API REST, utilisent des protocoles web standard, principalement HTTP, pour communiquer.

Les API REST sont conçues autour de ressources, qui sont des entités ou des objets de notre application web. Chaque ressource est accessible via des URLs uniques. Les interactions avec ces ressources se font à l'aide des méthodes HTTP (GET, POST, PUT, DELETE, etc.), permettant de réaliser des opérations CRUD (Créer, Lire, Mettre à jour, Supprimer).

Principes des API REST

Les API REST sont basées sur six principes fondamentaux qui définissent leur fonctionnement et leur structure :

- **Interface uniforme** : L'interaction avec une API REST doit être uniforme et standardisée, utilisant les méthodes HTTP de manière conventionnelle pour les opérations CRUD.
- **Sans état** (Stateless) : Chaque requête de l'API doit contenir toutes les informations nécessaires à son traitement. Le serveur ne doit pas garder de contexte ou d'état de session entre les requêtes.
- **Gestion des ressources par des représentations** : Les ressources sont manipulées en envoyant leurs représentations (souvent sous forme de JSON ou XML) entre le client et le serveur.
- **Ressources identifiables** : Les ressources sont identifiées dans les requêtes, généralement à l'aide d'URLs. Par exemple, une URL telle que /articles/1 peut représenter un article spécifique dans une application de blog.

Principes des API REST

- **Navigation hypertexte** (HATEOAS: Hypermedia As The Engine Of Application State) : Les réponses de l'API doivent inclure des liens hypertextes vers d'autres ressources associées, permettant au client de naviguer dynamiquement entre les états de l'application.

- **Système en couches** : Les clients communiquent avec une interface REST sans nécessairement savoir s'ils interagissent directement avec le serveur final ou avec un intermédiaire. Cela permet, entre autres, le déploiement de serveurs intermédiaires pour équilibrer la charge ou pour fournir un cache.

Pourquoi utiliser une API REST ?

Les API REST sont largement utilisées pour leur simplicité, leur flexibilité et leur facilité d'intégration avec les applications web et mobiles. Elles permettent de :

- **Découpler le frontend du backend**, facilitant le développement parallèle et l'évolution de chaque couche de manière indépendante.
- **Supporter différentes plateformes** et types de clients, de navigateurs web à des applications mobiles natives.
- **Simplifier les architectures** en utilisant le protocole HTTP standard pour toutes les opérations.
- **Faciliter la mise à l'échelle** des applications grâce à leur nature sans état.

2. Préparation de l'environnement

bstorm TRAINING

Installation de Django REST Framework

Django REST Framework (DRF) est une extension pour Django qui permet une construction aisée d'API RESTful. Pour l'installer :

- Installer Django REST Framework :

```
pip install djangorestframework
```

- Ajouter Django REST Framework à notre projet Django : Pour utiliser DRF, vous devez l'ajouter à la liste INSTALLED_APPS dans le fichier settings.py de notre projet Django :

```
INSTALLED_APPS = [
    ...
    'rest_framework',
]
```

Configuration de Base

Avec Django et Django REST Framework installés, effectuez les configurations initiales pour préparer le développement de notre API :

- Créer un projet Django : Si vous n'avez pas encore de projet, créez-en un en utilisant la commande :

```
django-admin startproject nom_de_projet
```

- Naviguez ensuite dans le répertoire du projet :

```
cd nom_de_projet
```

- Créer une application Django : Les projets Django sont composés d'applications, qui sont des modules Python servant à encapsuler des fonctionnalités spécifiques. Créez notre première application :

```
python manage.py startapp nom_de_app
```

N'oubliez pas d'ajouter notre application à INSTALLED_APPS dans settings.py.

Configurer la base de données

Django est livré avec une base de données SQLite par défaut, qui est suffisante pour les projets de petite à moyenne taille et pour l'apprentissage. Assurez-vous que les configurations de la base de données dans settings.py sont correctes. Si vous utilisez SQLite, la configuration par défaut devrait fonctionner sans problème.
Dans notre cas, nous allons utiliser PostegreSQL.

Pour ce faire nous avons besoin d'installer un package :

```
pip install psycopg2
```

Puis, configurez les paramètres de connexion à la base de données dans le fichier settings.py :

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'nom_de_votre_base_de_données',
        'USER': 'votre_utilisateur',
        'PASSWORD': 'votre_mot_de_passe',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

Finalement, il nous reste à effectuer une migration afin d'appliquer les changements :

```
python manage.py migrate
```

3. Modèle et Sérialisation

bstorm TRAINING

Définition du Modèle

Commencer par définir des modèles en Django, en particulier pour les APIs, repose sur l'efficacité de Django ORM, un outil puissant pour interagir avec la base de données via des objets Python. Cela permet une abstraction des opérations de base de données, rendant le code plus propre, plus sécurisé, et plus facile à maintenir. Les modèles servent de fondation solide pour la structure des données, sur laquelle on peut construire des fonctionnalités complexes avec moins de code et une meilleure intégration avec des outils comme Django REST Framework. En somme, commencer par les modèles capitalise sur les forces de Django pour un développement d'APIs rapide, sécurisé et maintenable

Définition du Modèle

```python
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    description = models.TextField()
    publication_date = models.DateField()
    genre = models.CharField(max_length=50)
    note = models.FloatField(default=0)

    def __str__(self):
        return self.title
```

Sérialisation

La sérialisation est le processus de transformation des données complexes telles que les objets Django QuerySet en un format qui peut être facilement rendu en JSON, XML ou d'autres types de description. DRF fournit un puissant système de sérialisation qui gère ces conversions pour vous.

Pour sérialiser le modèle Book, créez un fichier serializers.py dans notre application et définissez un BookSerializer :

```python
from rest_framework import serializers
from books.models import Book

class BookSerializer(serializers.Serializer):
    id = serializers.IntegerField(read_only=True)
    title = serializers.CharField(max_length=100)
    description = serializers.CharField()
    publication_date = serializers.DateField()
    genre = serializers.CharField(max_length=50)
    note = serializers.FloatField(default=0)
```

Sérialisation

```python
def create(self, validated_data):
    """
    Create and return a new `Book` instance, given the validated data.
    """
    return Book.objects.create(**validated_data)

def update(self, instance, validated_data):
    """
    Update and return an existing `Book` instance, given the validated data.
    """
    instance.title = validated_data.get('title', instance.title)
    instance.description = validated_data.get('description', instance.description)
    instance.publication_date = validated_data.get('publication_date', instance.publication_date)
    instance.genre = validated_data.get('genre', instance.genre)
    instance.note = validated_data.get('note', instance.note)
    instance.save()
    return instance
```

Sérialisation

La première partie de la classe sérialiseur définit les champs qui sont sérialisés/désérialisés. Les méthodes create() et update() définissent comment les instances pleinement constituées sont créées ou modifiées lors de l'appel à serializer.save()

Une classe sérialiseur est très similaire à une classe Form de Django, et inclut des indicateurs de validation similaires sur les différents champs, tels que required, max_length et default.

Nous pourrons gagner du temps en utilisant la classe ModelSerializer.

Travailler avec les sérialiseurs

Afin de comprendre le fonctionnement des sérialiseurs, nous pouvons travailler avec directement depuis l'interpréteur python.

Pour cela, nous allons utiliser le shell avec notre moteur django :

```
python manage.py shell
```

Nous pouvons ensuite importer notre modèle et notre sérialiseur.

```
from books.models import Book
from books.serializers import BookSerializer
```

Travailler avec les sérialiseurs

Nous allons également importer et utiliser JSONRenderer et JSONParser qui sont des composants qui facilitent la sérialisation et la désérialisation des données :

- JSONRenderer est utilisé pour convertir des structures de données Python en un format JSON. C'est utile lorsque vous voulez rendre la réponse de votre API en JSON, ce qui est le format standard pour la plupart des APIs web.

- JSONParser fait l'inverse ; il analyse les données JSON reçues dans une requête pour les convertir en types de données Python appropriés. Cela permet à votre API de traiter les données envoyées par le client.

```
from rest_framework.renderers import JSONRenderer
from rest_framework.parsers import JSONParser
```

Travailler avec les sérialiseurs

Créons des données à exploiter :

```
book = Book(title='Le Seigneur des Anneaux', description='Une épopée fantastique de J.R.R.
Tolkien.', publication_date='1954-07-29', genre='Fantasy', note=4.8)
book.save()

book = Book(title='1984', description='Un roman dystopique de George Orwell.',
publication_date='1949-06-08', genre='Dystopian Fiction', note=4.6)
book.save()
```

Nous avons maintenant quelques livres à manipuler. Jetons un coup d'œil à la sérialisation de l'une de ces instances.

Travailler avec les sérialiseurs

Sérialiser :

```
serializer = BookSerializer(book)
serializer.data
# {'id': 2, 'title': '1984', 'description': 'Un roman dystopique de George Orwell.',
'publication_date': '1949-06-08', 'genre': 'Dystopian Fiction', 'note': 4.6}
```

nous avons traduit l'instance du modèle en types de données natifs Python. Pour finaliser le processus de sérialisation, nous devons transformer les données en JSON.

```
content = JSONRenderer().render(serializer.data)
content
# b'{"id": 2, "title": "1984", "description": "Un roman dystopique de George Orwell.",
"publication_date": "1949-06-08", "genre": "Dystopian Fiction", "note": 4.6}'
```

Travailler avec les sérialiseurs

Désérialiser est similaire. Tout d'abord, nous analysons un flux en types de données natifs Python:

```
import io

stream = io.BytesIO(content)
data = JSONParser().parse(stream)
```

Ensuite, nous restaurons ces types de données natifs en une instance d'objet entièrement peuplée

```
serializer = BookSerializer(data=data)
serializer.is_valid()
# True
serializer.validated_data
# OrderedDict([('title', '1984'), ('description', 'Un roman dystopique de George Orwell.'),
('publication_date', '1949-06-08'), ('genre', 'Dystopian Fiction'), ('note', 4.6)])
serializer.save()
# <Book: Book object>
```

Travailler avec les sérialiseurs

Nous pouvons également sérialiser des querysets au lieu d'instances de modèle. Pour ce faire, nous ajoutons simplement un indicateur many=True aux arguments du sérialiseur.

```
serializer = BookSerializer(Book.objects.all(), many=True)
serializer.data
# [OrderedDict([('id', 1), ('title', 'Le Seigneur des Anneaux'), ('description', 'Une épopée
fantastique de J.R.R. Tolkien.'), ('publication_date', '1954-07-29'), ('genre', 'Fantasy'),
('note', 4.8)]), OrderedDict([('id', 2), ('title', '1984'), ('description', 'Un roman dystopique de
George Orwell.'), ('publication_date', '1949-06-08'), ('genre', 'Dystopian Fiction'), ('note',
4.6)])]
```

ModelSerializer

Le ModelSerializer dans Django REST Framework (DRF) est un outil puissant qui simplifie la création de sérialiseurs pour vos modèles Django. Il hérite de Serializer mais ajoute une couche d'abstraction qui automatise une grande partie de la configuration nécessaire pour transformer vos modèles en données JSON et inversement. Voici quelques points clés pour comprendre son utilisation et ses avantages :

- **Automatisation des Champs** : ModelSerializer génère automatiquement un ensemble de champs de sérialiseur basés sur le modèle donné. Pour chaque champ du modèle, il choisit un type de champ de sérialiseur correspondant. Cela réduit considérablement la quantité de code que vous devez écrire, surtout pour des modèles avec de nombreux champs.

- **Validation Automatique** : Il utilise les validations définies dans le modèle Django (telles que max_length pour un CharField) pour ajouter automatiquement ces validations aux champs du sérialiseur. Cela assure que les données fournies à l'API respectent les contraintes du modèle.

ModelSerializer

- **Création d'Instances** : ModelSerializer fournit une implémentation par défaut de la méthode create(), qui définit comment une nouvelle instance du modèle est créée à partir des données validées fournies à un sérialiseur.

- **Mise à Jour d'Instances** : De manière similaire, il implémente une méthode update() par défaut, qui gère la mise à jour d'une instance de modèle existante.

- **Personnalisation des Champs** : Bien que ModelSerializer génère automatiquement des champs, vous avez toujours la possibilité de les déclarer explicitement pour les personnaliser ou d'ajouter des champs qui n'existent pas sur le modèle.

- **Méta Classe** : La configuration du sérialiseur, telle que le modèle cible et les champs à inclure ou exclure, est définie dans une classe Meta interne, ce qui rend la configuration claire et concise.

ModelSerializer

```python
from rest_framework import serializers
from .models import Book

class BookSerializer(serializers.ModelSerializer):
    class Meta:
        model = Book
        fields = ['id', 'title', 'description', 'genre']
```

Dans cet exemple, BookSerializer générera automatiquement des sérialiseurs pour les champs id, title, description, et genre basés sur les définitions de champs dans le modèle Book. Les validations du modèle, telles que les longueurs maximales des champs de chaîne, seront appliquées automatiquement aux données entrantes lors de la sérialisation.

Si vous souhaitez inclure tous les champs d'un modèle dans un sérialiseur en utilisant Django REST Framework (DRF), vous pouvez le faire très simplement avec un ModelSerializer en utilisant l'option fields = 'all' dans la classe Meta de votre sérialiseur. Cela indique à DRF d'inclure automatiquement tous les champs du modèle dans le sérialiseur.

4. Vues et Urls

bstorm TRAINING

Introduction aux Vues et Urls

Les vues et les Urls constituent le cœur de l'interaction entre les données de votre application et le monde extérieur dans le cadre de Django REST Framework. Après avoir défini comment nos données doivent être sérialisées, nous allons maintenant exploiter pleinement cette configuration grâce aux vues, qui agissent comme des ponts entre les modèles de notre base de données et les requêtes HTTP entrantes. Les vues prennent en charge la logique métier, en déterminant comment les requêtes pour certaines URL doivent être traitées et quelles réponses doivent être renvoyées au client.

Travailler avec les Vues et Routeurs

Dans notre application Book, nous pouvons maintenant travailler avec views.py

Nous avons besoin de mettre en place les imports utilisés précédemment :

```
from django.http import HttpResponse, JsonResponse
from django.views.decorators.csrf import csrf_exempt
from rest_framework.parsers import JSONParser
from books.models import Book
from books.serializers import BookSerializer
```

La racine de notre API sera une vue qui prend en charge l'énumération de tous les livres existants, ou la création d'un nouveau livre.

nous voulons pouvoir POST à cette vue depuis des clients qui n'auront pas de jeton CSRF, nous devons marquer la vue comme csrf_exempt*.

**Ce n'est pas quelque chose que vous voudriez normalement faire, et les vues de REST framework utilisent en fait un comportement plus sensé que cela, mais cela fera l'affaire pour nos besoins actuels.*

```python
@csrf_exempt
def book_list(request):
    """
    List all books, or create a new book.
    """
    if request.method == 'GET':
        books = Book.objects.all()
        serializer = BookSerializer(books, many=True)
        return JsonResponse(serializer.data, safe=False)

    elif request.method == 'POST':
        data = JSONParser().parse(request)
        serializer = BookSerializer(data=data)
        if serializer.is_valid():
            serializer.save()
            return JsonResponse(serializer.data, status=201)
        return JsonResponse(serializer.errors, status=400)
```

Nous aurons également besoin d'une vue qui correspond à un livre individuel, et qui peut être utilisée pour récupérer, mettre à jour ou supprimer le livre.

Les vues déterminent la logique de gestion des requêtes HTTP.

DRF offre plusieurs manières de créer des vues pour une API, des vues basées sur des fonctions aux vues basées sur des classes, en passant par les viewsets pour une abstraction encore plus élevée.

```python
@csrf_exempt
def book_detail(request, pk):
    """
    Retrieve, update or delete a book.
    """
    try:
        book = Book.objects.get(pk=pk)
    except Book.DoesNotExist:
        return HttpResponse(status=404)

    if request.method == 'GET':
        serializer = BookSerializer(book)
        return JsonResponse(serializer.data)

    elif request.method == 'PUT':
        data = JSONParser().parse(request)
        serializer = BookSerializer(book, data=data)
        if serializer.is_valid():
            serializer.save()
            return JsonResponse(serializer.data)
        return JsonResponse(serializer.errors, status=400)

    elif request.method == 'DELETE':
        book.delete()
        return HttpResponse(status=204)
```

Status Code

Dans une API REST, l'utilisation des codes de statut HTTP est cruciale car elle fournit un mécanisme standardisé pour indiquer le résultat des requêtes HTTP entre le client et le serveur.
Chaque code de statut communique de manière concise et précise l'état de la requête, permettant ainsi aux développeurs de comprendre rapidement si une opération a réussi, échoué, ou nécessite une action supplémentaire. Cela simplifie le traitement des réponses côté client, puisque le code peut prendre des décisions basées sur le statut de la réponse, comme réessayer une requête, demander une authentification, ou gérer une erreur. En outre, l'utilisation des codes de statut HTTP contribue à l'uniformité et à l'interopérabilité entre différentes APIs et clients, facilitant l'intégration entre systèmes et services divers.
En somme, les codes de statut HTTP renforcent la clarté, l'efficacité et la robustesse des communications dans les architectures RESTful.

Status Code

- 200 OK : Indique que la requête a été traitée avec succès. Utilisé pour les réponses GET quand les données sont renvoyées sans problème.
- 201 Created : Signale qu'une nouvelle ressource a été créée en réponse à la requête. Souvent utilisé après une requête POST.
- 400 Bad Request : Utilisé quand le serveur ne peut pas traiter la requête en raison d'une erreur client, comme des données de requête mal formées.
- 404 Not Found : Signifie que la ressource demandée n'a pas été trouvée sur le serveur. Utilisé, par exemple, lorsqu'un ID de livre spécifié dans une requête GET n'existe pas.
- 204 No Content : Indique que la requête a été traitée avec succès mais qu'il n'y a pas de contenu à renvoyer. Typiquement utilisé pour les requêtes DELETE.
- 401 Unauthorized : Signale que la requête nécessite une authentification. Utilisé quand l'accès à une ressource est restreint et nécessite des credentials.
- 403 Forbidden : Indique que le serveur refuse d'exécuter la requête, même si l'utilisateur est authentifié. Utilisé pour contrôler l'accès aux ressources.
- 405 Method Not Allowed : Utilisé pour indiquer que la méthode HTTP utilisée n'est pas autorisée pour la ressource demandée. Par exemple, si une requête POST est faite sur une URL qui n'accepte que les requêtes GET.
- 500 Internal Server Error : Signifie que le serveur a rencontré une condition inattendue qui l'a empêché de répondre à la requête. Utilisé quand une erreur serveur empêche l'exécution de la requête.

patterns d'URL

Pour que les requêtes HTTP atteignent vos vues, vous devez configurer les URLs dans notre application Django. DRF facilite cette tâche avec un système de routage simple et puissant.

Les patterns d'URL sont définis dans le fichier urls.py de chaque application Django, en utilisant la fonction path() pour associer des chemins d'URL à des vues spécifiques. Les vues peuvent alors utiliser les paramètres capturés pour effectuer des opérations telles que récupérer, mettre à jour ou supprimer des ressources basées sur leur identifiant.

```
from django.urls import path
from books import views

urlpatterns = [
    path('books/', views.book_list),
    path('books/<int:pk>/', views.book_detail),
]
```

Paramètres d'URL

Dans le développement web avec Django, y compris avec Django REST Framework (DRF), la définition des routes et la capture de paramètres dans les URLs sont essentielles pour créer des applications web dynamiques et des APIs RESTful.

Django permet donc de spécifier des types de paramètres dans les chemins d'URL à l'aide de convertisseurs :

<int:pk> : Capture un entier et le passe à la vue comme une clé primaire (pk).

<slug:slug> : Capture une chaîne formatée comme un slug (URL SEO-friendly) et la passe à la vue.

<str:name> : Capture toute chaîne non vide.

<uuid:uuid> : Capture un UUID.

<path:path> : Capture un chemin complet, y compris les slashs /.

Paramètres d'URL

La spécification de paramètres d'URL avec des convertisseurs permet de :

- Valider et capturer des données directement à partir de l'URL.
- Créer des URL descriptives et faciles à comprendre.
- Faciliter l'accès et la manipulation de ressources spécifiques dans les applications web et les APIs.

Wrapping dans Django REST Framework

DRF propose deux méthodes principales pour "envelopper" les vues de notre API : le décorateur @api_view pour les fonctions et la classe APIView pour les vues basées sur des classes. Ces méthodes enrichissent vos vues Django standard avec des fonctionnalités supplémentaires spécifiques aux APIs, telles que la négociation de contenu, le parsing des requêtes, et la gestion uniforme des réponses.

- **Le décorateur @api_view** vous permet de transformer une vue basée sur une fonction en une vue qui peut gérer des requêtes RESTful. Vous spécifiez les méthodes HTTP que la vue doit accepter (comme GET ou POST) et DRF s'occupe du reste.

- **La classe APIView** offre une approche orientée objet qui agit comme une base pour vos vues. Elle fournit une structure claire pour gérer différentes méthodes HTTP et intègre des fonctionnalités supplémentaires pour la gestion des requêtes et des réponses.

Utilisation du décorateur @api_view

```python
from rest_framework.decorators import api_view
from rest_framework.response import Response
from .serializers import BookSerializer
from .models import Book

@api_view(['GET', 'POST'])
def book_list(request):
    """
    List all books, or create a new book.
    """
    if request.method == 'GET':
        books = Book.objects.all()
        serializer = BookSerializer(books, many=True)
        return Response(serializer.data)

    elif request.method == 'POST':
        serializer = BookSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=201)
        return Response(serializer.errors, status=400)
```

Utilisation de la classe APIView

```python
from rest_framework.views import APIView
from rest_framework.response import Response
from .serializers import BookSerializer
from .models import Book

class BookList(APIView):
    """
    List all books, or create a new book.
    """
    def get(self, request, format=None):
        books = Book.objects.all()
        serializer = BookSerializer(books, many=True)
        return Response(serializer.data)

    def post(self, request, format=None):
        serializer = BookSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=201)
        return Response(serializer.errors, status=400)
```

Wrapping dans Django REST Framework

Après avoir mis en place nos vues avec le wrapping, il est temps de les intégrer à nos URLs, afin qu'elles puissent répondre concrètement aux requêtes de notre API. Cette étape nécessite de convertir nos vues, actuellement définies comme des classes, en vues fonctionnelles que Django peut reconnaître et gérer.

C'est ici que la méthode .as_view() va être utilisée, faisant le lien nécessaire entre nos vues orientées objet et les attentes de Django pour le traitement des requêtes.

```
from django.urls import path
from .views import BookDetail

urlpatterns = [
    path('books/<int:pk>/', BookDetail.as_view(), name='book-detail'),
]
```

BookDetail.as_view() transforme notre classe de vue en une vue fonctionnelle, prête à être utilisée par Django. Cela assure que les méthodes définies dans notre classe sont correctement invoquées en réponse aux requêtes HTTP.

La propriété format

Dans l'utilisation de la classe APIView vous remarquerez la propriété format, celle-ci est utilisée pour déterminer le format de la réponse que l'API doit renvoyer. Elle permet à une même vue de gérer différentes représentations des données, telles que JSON, HTML, ou d'autres formats personnalisés. Cela est particulièrement utile pour construire des APIs flexibles qui peuvent répondre aux besoins de divers clients ou consommateurs de l'API.

Lorsque vous incluez format=None dans les signatures de méthodes d'une vue basée sur APIView, vous permettez explicitement à cette vue de gérer les suffixes de format spécifiés dans l'URL de la requête. Par exemple, si un client fait une requête à /api/books/1.json, DRF comprend que la réponse doit être formatée en JSON. Si le client utilise /api/books/1.api, la réponse pourrait être en HTML ou dans un autre format spécifié par le système de négociation de contenu de DRF.

La propriété format

L'utilisation de la propriété format offre plusieurs avantages :

- **Flexibilité** : Elle permet à notre API de servir les données dans le format préféré du client, augmentant ainsi la compatibilité et la facilité d'utilisation de l'API.

- **Simplicité** : En gérant différents formats à un seul et même endroit, cela simplifie la conception de vos vues et centralise la logique de formatage.

- **Négociation de Contenu** : DRF utilise un système de négociation de contenu pour déterminer automatiquement le meilleur format à utiliser pour une réponse, basé sur la requête du client. La propriété format joue un rôle clé dans ce processus en fournissant une indication explicite du format demandé.

5. Authentification et Permissions

bstorm TRAINING

Authentification

Dans le développement d'API REST avec Django REST Framework (DRF), la gestion de l'authentification et des permissions est essentielle pour sécuriser l'accès aux ressources et s'assurer que les utilisateurs ont les droits appropriés pour effectuer des actions spécifiques.

L'authentification détermine l'identité de l'utilisateur qui effectue une requête à l'API. DRF supporte plusieurs schémas d'authentification, dont les plus courants sont :

- **Basic Authentication** : Simple mais peu sécurisée, cette méthode transmet le nom d'utilisateur et le mot de passe en clair avec chaque requête. Elle est principalement utilisée pour le développement et le test.
- **Token Authentication** : Une clé secrète (token) est utilisée pour authentifier les requêtes. Le token est obtenu via une requête d'authentification et doit être inclus dans les en-têtes des requêtes suivantes.
- **JWT (JSON Web Token) Authentication** : Similaire à la Token Authentication, mais utilise des JSON Web Tokens, qui peuvent inclure des informations supplémentaires sur l'utilisateur et sont sécurisés contre les altérations.

La gestion des JWT

La gestion des JWT (JSON Web Tokens) dans Django REST Framework nécessite une configuration supplémentaire, car DRF ne supporte pas les JWT dans sa configuration par défaut. Cependant, grâce à des packages tiers comme djangorestframework-simplejwt, l'intégration de JWT devient assez simple. Voici comment configurer et utiliser JWT pour l'authentification dans notre projet Django.

Installez djangorestframework-simplejwt en utilisant pip :

pip install djangorestframework-simplejwt

```
pip install djangorestframework-simplejwt
```

La gestion des JWT

Ajoutez rest_framework_simplejwt.authentication.JWTAuthentication à DEFAULT_AUTHENTICATION_CLASSES dans vos paramètres DRF :

```
INSTALLED_APPS = [
    ...
    'rest_framework',
    'rest_framework_simplejwt',
    ...
]

REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ),
}
```

La gestion des JWT

djangorestframework-simplejwt fournit des vues pour obtenir et rafraîchir les tokens. Vous devez inclure ces vues dans vos URLs.

```
from django.urls import path
from rest_framework_simplejwt.views import (
    TokenObtainPairView,
    TokenRefreshView,
)

urlpatterns = [
    path('api/token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),
    path('api/token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),
]
```

Ces endpoints vous permettent d'obtenir un token JWT en fournissant un nom d'utilisateur et un mot de passe, ainsi que de rafraîchir ce token une fois qu'il est proche de l'expiration.

La gestion des JWT

Pour obtenir un token, envoyez une requête POST à /api/token/ avec un nom d'utilisateur et un mot de passe :

```
{
  "username": "votre_nom_d_utilisateur",
  "password": "votre_mot_de_passe"
}
```

La réponse contiendra un access token et un refresh token.

Pour accéder à une vue protégée, incluez le token d'accès dans l'en-tête d'autorisation de votre requête HTTP :

```
Authorization: Bearer votre_token_jwt
```

La gestion des JWT

Lorsque le token d'accès est proche de l'expiration, vous pouvez obtenir un nouveau token d'accès en envoyant le refresh token à /api/token/refresh/.

Sécurité et Pratiques Recommandées

- **Stockage des Tokens** : Côté client, stockez les tokens JWT de manière sécurisée pour prévenir les fuites de données.
- **HTTPS** : Utilisez toujours HTTPS pour communiquer avec votre API afin de protéger les tokens et les données sensibles contre les interceptions.
- **Personnalisation** : djangorestframework-simplejwt permet de personnaliser le comportement des tokens, comme leur durée de vie. Consultez la documentation pour ajuster ces paramètres selon vos besoins.

Permissions

es permissions contrôlent si un utilisateur authentifié a le droit d'effectuer une action (comme lire, modifier ou supprimer une ressource). DRF fournit un ensemble de classes de permission prédéfinies et permet également de créer des permissions personnalisées.

Classes de Permission Courantes :

- AllowAny : Autorise toutes les requêtes, qu'elles soient authentifiées ou non.
- IsAuthenticated : N'autorise que les requêtes d'utilisateurs authentifiés.
- IsAdminUser* : Restreint l'accès aux utilisateurs ayant le statut d'administrateur.
- IsAuthenticatedOrReadOnly : Autorise les requêtes en lecture à tout le monde, mais restreint les actions de modification aux utilisateurs authentifiés.

**Afin d'utiliser IsAdminUser il faut ajouter le rôle dans le payload dans le cas de l'utilisation d'un JWT*

Permissions

Pour utiliser une classe de permission dans une vue, utilisez l'attribut permission_classes

Grâce à la persmission courante [IsAuthenticated], toutes les méthodes à l'intérieur de la classe BookDetail seront autorisées que pour les membres connectés.

Pour appliquer la permission sur une seule méthode nous utiliserons alors à l'intérieur de la méthode :

```
if request.user.is_authenticated:
   #Instructions
```

```python
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.permissions import IsAuthenticated
from .models import Book
from .serializers import BookSerializer
from django.http import Http404

class BookDetail(APIView):
    permission_classes = [IsAuthenticated]

    def get_object(self, pk):
        try:
            return Book.objects.get(pk=pk)
        except Book.DoesNotExist:
            raise Http404

    def get(self, request, pk, format=None):
        book = self.get_object(pk)
        serializer = BookSerializer(book)
        return Response(serializer.data)
```

6. Relations et Hyperliens

bstorm TRAINING

Relations et Hyperliens

Dans une API REST construite avec Django REST Framework (DRF), gérer les relations entre différents modèles de données est essentiel. DRF facilite la représentation des relations entre les entités — dans ce cas, entre des livres et d'autres entités — et nous verrons comment utiliser les hyperliens pour naviguer entre les ressources liées dans notre API.

Types de Relations

- ForeignKey (Many-to-One) : Un livre peut avoir un seul auteur, mais un auteur peut écrire plusieurs livres.
- ManyToManyField (Many-to-Many) : Un livre peut appartenir à plusieurs catégories, et une catégorie peut contenir plusieurs livres.
- OneToOneField (One-to-One) : Chaque livre peut avoir une seule instance de méta-données spécifiques.

Représentation des Relations

Pour représenter ces relations dans vos sérialiseurs, DRF offre plusieurs options :

- PrimaryKeyRelatedField : Utilise l'ID de l'entité liée.
- HyperlinkedRelatedField : Utilise les URL comme moyen de représenter les relations, favorisant une navigation API basée sur des hyperliens.
- StringRelatedField : Utilise la représentation en chaîne du modèle lié (définie par la méthode __str__ du modèle).
- SlugRelatedField : Utilise un champ slug du modèle lié pour représenter la relation.
- Nested Relationships : Représente les relations en imbriquant les sérialiseurs.

Utilisation des Hyperliens

L'utilisation des hyperliens offre plusieurs avantages :

- **Découvrabilité** : Les utilisateurs de votre API peuvent naviguer entre les ressources liées en suivant les URL, améliorant ainsi la découvrabilité de l'API.

- **Cohérence** : Les URL fournissent un mécanisme cohérent et standard pour référencer les ressources, contrairement aux ID ou aux slugs qui peuvent varier en format.

- **Facilité de Maintenance** : Les changements dans la structure de l'API n'affectent pas les consommateurs tant que les URL restent les mêmes.

Mise en place des Hyperliens

Imaginons que vous souhaitiez lier des auteurs à des livres. Voici comment vous pourriez structurer votre modèle et sérialiseur :

```python
# models.py
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)
    bio = models.TextField()

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey(Author, related_name='books', on_delete=models.CASCADE)
    summary = models.TextField()
```

BookSerializer utilise ici HyperlinkedModelSerializer pour inclure des hyperliens vers les auteurs. De même, AuthorSerializer inclut des liens vers tous les livres écrits par l'auteur.

```python
# serializers.py
from rest_framework import serializers
from .models import Book, Author

class BookSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Book
        fields = ['url', 'title', 'author', 'summary']
        extra_kwargs = {
            'author': {'view_name': 'author-detail', 'lookup_field': 'pk'}
        }

class AuthorSerializer(serializers.HyperlinkedModelSerializer):
    books = serializers.HyperlinkedRelatedField(many=True, view_name='book-detail', read_only=True)

    class Meta:
        model = Author
        fields = ['url', 'name', 'bio', 'books']
```

7. Vues Avancées et Routage

bstorm
TRAINING

Introduction

Les vues génériques et les ViewSets sont des composants puissants et flexibles dans le développement d'applications web avec des frameworks comme Django et Django REST Framework (DRF). Ils permettent de simplifier et d'accélérer le processus de développement en fournissant des solutions prêtes à l'emploi pour des tâches courantes telles que la manipulation des données, la gestion des formulaires, l'authentification, etc.

Avantages :

- **Réutilisabilité** : Les vues génériques et les ViewSets permettent de réduire la duplication de code en fournissant des solutions prêtes à l'emploi pour des tâches courantes.
- **Rapidité de développement** : En utilisant des composants pré-construits, les développeurs peuvent accélérer le processus de développement et se concentrer sur la logique métier spécifique de l'application.
- **Maintenabilité** : Les vues génériques et les ViewSets suivent les meilleures pratiques de conception et sont conçus pour être extensibles et maintenables à long terme.

Vues Génériques (Generic Views) :

- Les vues génériques sont des classes prédéfinies fournies par Django pour effectuer des opérations courantes sur les données.
- Elles offrent une implémentation générique pour des actions telles que l'affichage de listes d'objets, la création, la mise à jour et la suppression d'objets.
- Les vues génériques sont conçues pour être réutilisables, ce qui permet de réduire la duplication de code et d'améliorer la maintenabilité du code.
- Elles sont configurables via des attributs de classe et des méthodes de classe, ce qui permet d'adapter leur comportement aux besoins spécifiques de l'application.

ViewSets :

- Les ViewSets sont une fonctionnalité clé de Django REST Framework (DRF) pour la création d'API REST.
- Ils regroupent un ensemble de vues liées à un modèle ou à une ressource particulière et fournissent des fonctionnalités CRUD (Create, Read, Update, Delete) pour cette ressource.
- Les ViewSets sont conçus pour fonctionner avec les routeurs de DRF, ce qui simplifie la configuration des URL pour les API.
- Ils sont souvent utilisés en conjonction avec les routeurs et les sérialiseurs de DRF pour construire rapidement des API RESTful conformes aux meilleures pratiques.

Vues Génériques (Generic Views) :

DRF fournit des vues génériques qui pré-implémentent certaines logiques pour les opérations CRUD (Créer, Lire, Mettre à jour, Supprimer). Utiliser ces vues peut simplifier votre code de vue.

```python
from rest_framework import generics
from .models import Book
from .serializers import BookSerializer

class BookListCreate(generics.ListCreateAPIView):
    queryset = Book.objects.all()
    serializer_class = BookSerializer

class BookDetailUpdateDelete(generics.RetrieveUpdateDestroyAPIView):
    queryset = Book.objects.all()
    serializer_class = BookSerializer
```

Ici, BookListCreate fournit automatiquement les actions pour lister et créer des livres, tandis que BookDetailUpdateDelete permet de récupérer, mettre à jour ou supprimer un livre spécifique.

ViewSets :

Pour simplifier davantage, DRF propose les ViewSets qui regroupent la logique de différentes vues en une seule classe.

```python
from rest_framework import viewsets
from .models import Book
from .serializers import BookSerializer

class BookViewSet(viewsets.ModelViewSet):
    queryset = Book.objects.all()
    serializer_class = BookSerializer
```

BookViewSet combine les opérations de listage, création, récupération, mise à jour et suppression en une classe. Cela rend le routage plus simple et plus centralisé.

Vue Générique	Description
APIView	Vue de base pour la création de vues personnalisées. Vous devez définir la logique de gestion des requêtes HTTP (GET, POST, PUT, DELETE, etc.) dans les méthodes correspondantes telles que get(), post(), put(), delete(), etc.
GenericAPIView	Une vue générique qui fournit une implémentation de base pour les vues génériques. Elle inclut des fonctionnalités telles que la gestion de l'authentification, de la pagination et de la validation des données. Elle est souvent utilisée comme classe de base pour les autres vues génériques.
ListAPIView	Affiche une liste d'objets. Cette vue prend en charge les requêtes HTTP GET et fournit une pagination par défaut pour les résultats.
CreateAPIView	Crée un nouvel objet. Cette vue prend en charge les requêtes HTTP POST et valide les données soumises avant de les enregistrer dans la base de données.
RetrieveAPIView	Récupère les détails d'un objet individuel. Cette vue prend en charge les requêtes HTTP GET et utilise une clé primaire spécifiée dans l'URL pour identifier l'objet à récupérer.
UpdateAPIView	Met à jour un objet existant. Cette vue prend en charge les requêtes HTTP PUT et PATCH pour mettre à jour les données d'un objet existant.
DestroyAPIView	Supprime un objet existant. Cette vue prend en charge les requêtes HTTP DELETE pour supprimer un objet de la base de données.
ListCreateAPIView	Combine les fonctionnalités de ListAPIView et CreateAPIView. Cette vue prend en charge les requêtes HTTP GET pour afficher une liste d'objets et les requêtes HTTP POST pour créer de nouveaux objets.
RetrieveUpdateAPIView	Combine les fonctionnalités de RetrieveAPIView et UpdateAPIView. Cette vue prend en charge les requêtes HTTP GET pour récupérer les détails d'un objet et les requêtes HTTP PUT et PATCH pour mettre à jour les données de cet objet.
RetrieveDestroyAPIView	Combine les fonctionnalités de RetrieveAPIView et DestroyAPIView. Cette vue prend en charge les requêtes HTTP GET pour récupérer les détails d'un objet et les requêtes HTTP DELETE pour supprimer cet objet de la base de données.
RetrieveUpdateDestroyAPIView	Combine les fonctionnalités de RetrieveAPIView, UpdateAPIView et DestroyAPIView. Cette vue prend en charge les requêtes HTTP GET pour récupérer les détails d'un objet, les requêtes HTTP PUT et PATCH pour mettre à jour les données de cet objet, et les requêtes HTTP DELETE pour supprimer cet objet de la base de données.

Introduction au Routage avec les ViewSets

Les ViewSets dans Django REST Framework représentent une abstraction puissante qui regroupe les comportements liés à la logique des actions standards sur les modèles, tels que lire, écrire et mettre à jour des données. En combinant les ViewSets avec le système de routage de DRF, les développeurs peuvent bénéficier d'une automatisation accrue, ce qui simplifie la configuration des URL et améliore l'organisation du code.

Avantages du Routage Automatisé avec DefaultRouter

- Automatisation des Routes : Le DefaultRouter de DRF génère automatiquement les routes nécessaires pour toutes les actions d'un ViewSet, réduisant ainsi le besoin de définir manuellement chaque route pour les opérations CRUD (Créer, Lire, Mettre à jour, Supprimer). Cela simplifie la configuration des URL et assure que chaque ViewSet dispose d'une interface cohérente et prévisible.
- Réutilisabilité : En standardisant la création de routes, les DefaultRouter permettent une réutilisation accrue du code. Les développeurs peuvent appliquer le même ViewSet à différents endroits de leur application sans réécrire la logique de routage, assurant ainsi une meilleure cohérence et réduisant les erreurs.

- Rapidité de développement : La capacité de générer rapidement des routes pour des ViewSets complets accélère significativement le processus de développement. Les développeurs peuvent se concentrer sur la définition des comportements spécifiques aux données et des règles métier plutôt que sur la configuration des routes.
- Maintenabilité : Les routes générées par DefaultRouter sont maintenues dans un style cohérent et suivent les meilleures pratiques REST, ce qui rend l'application plus facile à comprendre et à maintenir. Les changements dans les ViewSets se reflètent automatiquement dans les routes sans intervention manuelle supplémentaire.

```python
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from .views import BookViewSet

router = DefaultRouter()
router.register(r'books', BookViewSet)

urlpatterns = [
    path('', include(router.urls)),
]
```

8. Filtrage, Pagination et Tri

bstorm TRAINING

Introduction

Lors de la construction d'APIs RESTful avec Django REST Framework (DRF), il est essentiel de fournir des moyens efficaces pour filtrer, paginer et trier les données. Ces fonctionnalités améliorent l'expérience utilisateur en facilitant la navigation dans les grandes collections de données, telles que votre collection de livres dans une API de gestion des livres (book).

Pagination

La pagination est cruciale pour gérer les performances et l'expérience utilisateur lors de l'interaction avec de grandes quantités de données. Elle permet de diviser les résultats en "pages" de données, réduisant ainsi la charge sur le réseau et le serveur.

Configurer la Pagination :

DRF propose plusieurs styles de pagination. Voici comment configurer une pagination simple :

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 10
}
```

Cette configuration dans settings.py indique à DRF d'utiliser une pagination par numéro de page avec 10 livres par page.

Travailler avec les filtres

Le filtrage permet aux utilisateurs de votre API de restreindre l'ensemble des données renvoyées selon certains critères. Par exemple, filtrer les livres par genre, auteur ou date de publication.

Pour utiliser les filtres dans DRF, nous devons installer django_filter via pip :

```
pip install django-filter
```

Ajouter django_filters à la liste INSTALLED_APPS dans votre fichier settings.py :

```
INSTALLED_APPS = [
    ...
    'django_filters',
    ...
]
```

Travailler avec les filtres

Pour configurer Django REST Framework et utiliser django_filters comme backend de filtrage pour Django REST Framework, vous devez également l'ajouter à la configuration de DRF dans votre settings.py. Assurez-vous que django_filters.rest_framework.DjangoFilterBackend est spécifié dans DEFAULT_FILTER_BACKENDS :

```
REST_FRAMEWORK = {
    ...
    'DEFAULT_FILTER_BACKENDS':
['django_filters.rest_framework.DjangoFilterBackend'],
    ...
}
```

Filtrage

Pour un filtrage simple basé sur les requêtes GET, DRF permet de surcharger la méthode get_queryset dans vos vues

```
from django_filters.rest_framework import DjangoFilterBackend
from rest_framework import generics
from .models import Book
from .serializers import BookSerializer

class BookList(generics.ListAPIView):
    queryset = Book.objects.all()
    serializer_class = BookSerializer
    filter_backends = [DjangoFilterBackend]
    filterset_fields = ['genre', 'author']
```

Ce code permet de filtrer les livres par genre et author en utilisant des paramètres de requête, par exemple /books/?genre=fantasy&author=J.R.R. Tolkien

Tri

Le tri permet aux utilisateurs de spécifier l'ordre dans lequel les résultats doivent être retournés. Par exemple, trier les livres par titre ou date de publication.

DRF intègre le tri par défaut via le OrderingFilter. Voici comment l'activer pour votre vue BookList :

```python
from rest_framework.filters import OrderingFilter

class BookList(generics.ListAPIView):
    queryset = Book.objects.all()
    serializer_class = BookSerializer
    filter_backends = [DjangoFilterBackend, OrderingFilter]
    filterset_fields = ['genre', 'author']
    ordering_fields = ['title', 'publication_date']
```

Cela permet aux utilisateurs de trier les livres par title ou publication_date en utilisant le paramètre de requête ordering, par exemple /books/?ordering=publication_date.

Merci pour votre attention.

bstorm
TRAINING

Source

DJANGO (MVT)

- Documentation officiel Django : https://docs.djangoproject.com/en/5.0/
- Apprendre la programmation web avec Python et Django:

  Principes et bonnes pratiques pour les sites web dynamiques
  de Gilles Degols (Auteur), Pierre Alexis (Auteur), Hugues Bersini (Auteur)

DJANGO (REST API)

- Django Rest Framework : https://www.django-rest-framework.org/